ARL-TR-7561 ● JAN 2016

US Army Research Laboratory

US Army Research Laboratory Visualization Framework Design Document

by Will Dron, Mark Keaton, John Hancock, Mathew Aguirre, and Andrew J Toth

ARL-TR-7561 ● JAN 2016

US Army Research Laboratory

US Army Research Laboratory Visualization Framework Design Document

by Will Dron and Mark Keaton
Raytheon BBN Technologies, 10 Moulton St, Cambridge, MA 02138

Andrew J Toth
Computation and Information Sciences Directorate, ARL

John Hancock and Mathew Aguirre
ArtisTech, 10560 Main Street, Suite 105, Fairfax, VA 22030

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE *(DD-MM-YYYY)* January 2016	**2. REPORT TYPE** Final	**3. DATES COVERED (From - To)** 10/2014–09/2015

4. TITLE AND SUBTITLE US Army Research Laboratory Visualization Framework Design Document	**5a. CONTRACT NUMBER**
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Will Dron, Mark Keaton, John Hancock, Mathew Aguirre, and Andrew J Toth	**5d. PROJECT NUMBER**
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) US Army Research Laboratory ATTN: RDRL-CIN-T 2800 Powder Mill Road Adelphi, MD 20783-1138	**8. PERFORMING ORGANIZATION REPORT NUMBER** ARL-TR-7561
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	**10. SPONSOR/MONITOR'S ACRONYM(S)**
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
Visualization of network science experimentation results is generally achieved using stovepipe solutions tailored to specific experiments and performance metrics. Based on ZeroMQ, the US Army Research Laboratory (ARL) Visualization Framework presents a language-agnostic, platform-independent approach to connecting data published by probes to visualizations using a publish/subscribe mechanism. The framework provides for automated discovery of probes by the visualization without prior knowledge of the probes. This report documents the ARL Visualization Framework system design and specific details of its implementation.

15. SUBJECT TERMS
Network Science, Visualization

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** Andrew J Toth
a. REPORT Unclassified	**b. ABSTRACT** Unclassified	**c. THIS PAGE** Unclassified	UU	40	**19b. TELEPHONE NUMBER (Include area code)** 301-394-2746

Standard Form 298 (Rev. 8/98)
Prescribed by ANSI Std. Z39.18

Contents

List of Figures

List of Tables

INTENTIONALLY LEFT BLANK.

1. Introduction

The US Army Research Laboratory's (ARL) Network Science Research Laboratory (NSRL) is composed of a suite of hardware and software that models the operation of mobile networked device radio frequency (RF) links through emulation (not merely simulation) (Fig. 1). NSRL enables experimental validation or falsification of theoretical models, and characterization of protocols and algorithms for mobile wireless networks. It is used for a range of experiments, from assessing in-network aggregation of network information for detecting cyber threats, to characterizing the impact of communications disruption on perceived trust and quality of information metrics delivered to Soldiers in tactical mobile environments. Unlike other experimentation facilities for research in wireless networks, NSRL is focused on Army-unique requirements like hybrid networks and extensive modeling of ground and urban effects on communications. NSRL supports investigation of traditional wireless networking challenges as well as more general network science research issues. The NSRL's emulation environment is result of collaborative efforts between ARL and the US Naval Research Laboratory (NRL).

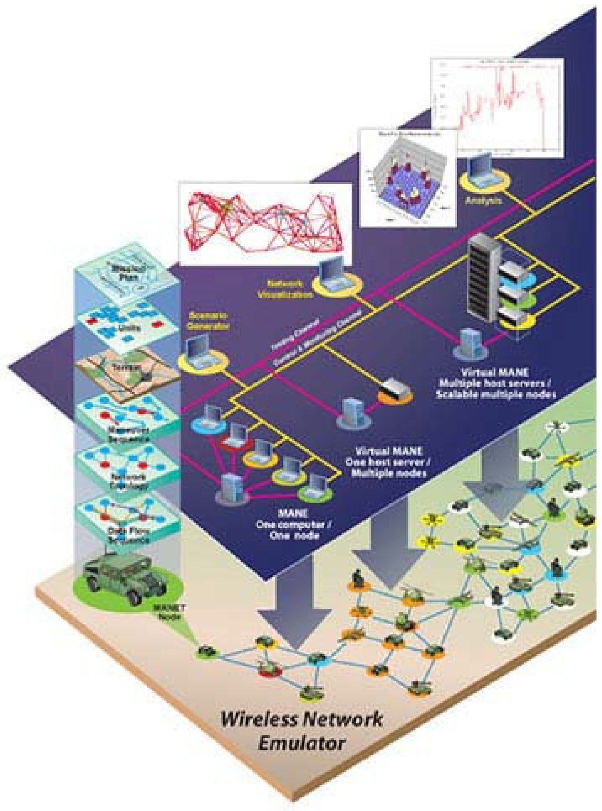

Fig. 1 NSRL

The software and systems running on the emulated networks execute in real time unlike simulations, which typically execute faster than real time by jumping from event to event skipping the time between events. One of the biggest advantages of emulated environments is that the characteristics of the network and its behaviors can be fully controlled and are repeatable. This is particularly interesting when emulating wireless networks as reproducibility of experimental environments using real radios is often difficult as temperature and humidity changes, differences in seasonal foliage, and other factors can alter the performance of the wireless networks. The primary emulation tools used by ARL are the Extendable Mobile Ad hoc Network Emulator (EMANE)[1] and the Common Open Research Emulator (CORE).[2]

Researchers at the NSRL facility identified a need for a suite of visualization capabilities to use to monitor experiments, view results, and provide a collaborative environment for researchers. This project integrates a variety of visualization tools in single and multi-machine environments. For distributed environments, multiple users will be able to control the whole system or parts of it, depending on how machines are allocated. The design of this system makes every effort to remain general enough that other visualization tools can be easily integrated into the US Army Research Laboratory Visualization Framework (ARL-VF) with minimal effort.

As this project continues, we are taking advantage of patterns commonly found in datasets and reuse them to push different types of data to the same tool. For example, latitude and longitude coordinates can be plotted into a network overview, 2- and 3-dimensional data points can be plotted in a graph, and tree-like structures can be plotted into an expanding pie chart. Additionally, we investigate different methods for storing data, using databases for cached data and TestPoint[3] to provide live data. Finally, we look carefully at common methods of formatting data to better compare experiment runs that range a wide variety of topics, such as communication networks, big dataset analysis, social dynamics, and other Army-relevant topics.

2. Goals and Requirements

2.1 Goals for Ease of Use

A computer savvy researcher who has never used this framework should be able to install, set up, and use the system within an hour. Within another hour, they should be able to port a simple dataset into the framework and view the results.

Non-computer-savvy researchers should be able to use a preinstalled system within an hour with some rudimentary understanding of how to visualize their own datasets.

It should be as easy or only trivially more effort to use the ARL-VF on a standalone system instead of using each visualization tool directly.

2.2 Language Agnostic

We expect visualization components to be written in a variety of different languages. Some may be run remotely, others must be run locally.

2.3 Operating System Agnostic

We do not expect every visualization tool incorporated into the ARL-VF to be compatible with every operating system. However, effort should be made to use operating system (OS)-independent tools. In cases where one tool is incompatible with a given OS, the framework must notify the user that such functionality is unavailable or offer a different tool for visualizing the data.

2.4 Environment

The system must be able to handle running on a single machine or distributed among several machines. These machines may be physically far apart; however, optimizing transfer of traffic for systems not in the same physical lab is beyond the scope of this design.

The system should be lightweight and require as few non-standard libraries as possible. Where non-standard libraries are used, such as ZeroMQ, it is noted in this report.

The system must be able to handle simultaneous input from multiple sources and use a locking/synchronization scheme to ensure there are no race conditions when 2 inputs send conflicting requests.

3. System Overview

The ARL-VF will use a ZeroMQ-based bus to route data from probes to VizDaemons interested in displaying the probe data using a publish/subscribe architecture. Probes can be any application or script that can produce ZeroMQ-compliant messages containing data of interest. These data can range from network metrics collected in an experiment to server central processing unit (CPU) statistics. VizDaemons consume only the ZeroMQ messages published by the probes to which they subscribe and display them accordingly. While VizDaemons are typically used for visual output types, there is no reason the output cannot be aural.

Figure 2 depicts an instance of the ARL-VF components in gray alongside TestPoint probes in purple. Both systems use a ZeroMQ message bus with similar protocol buffers. This section highlights each module in the ARL-VF and subsequent sections provide details on how each module interacts.

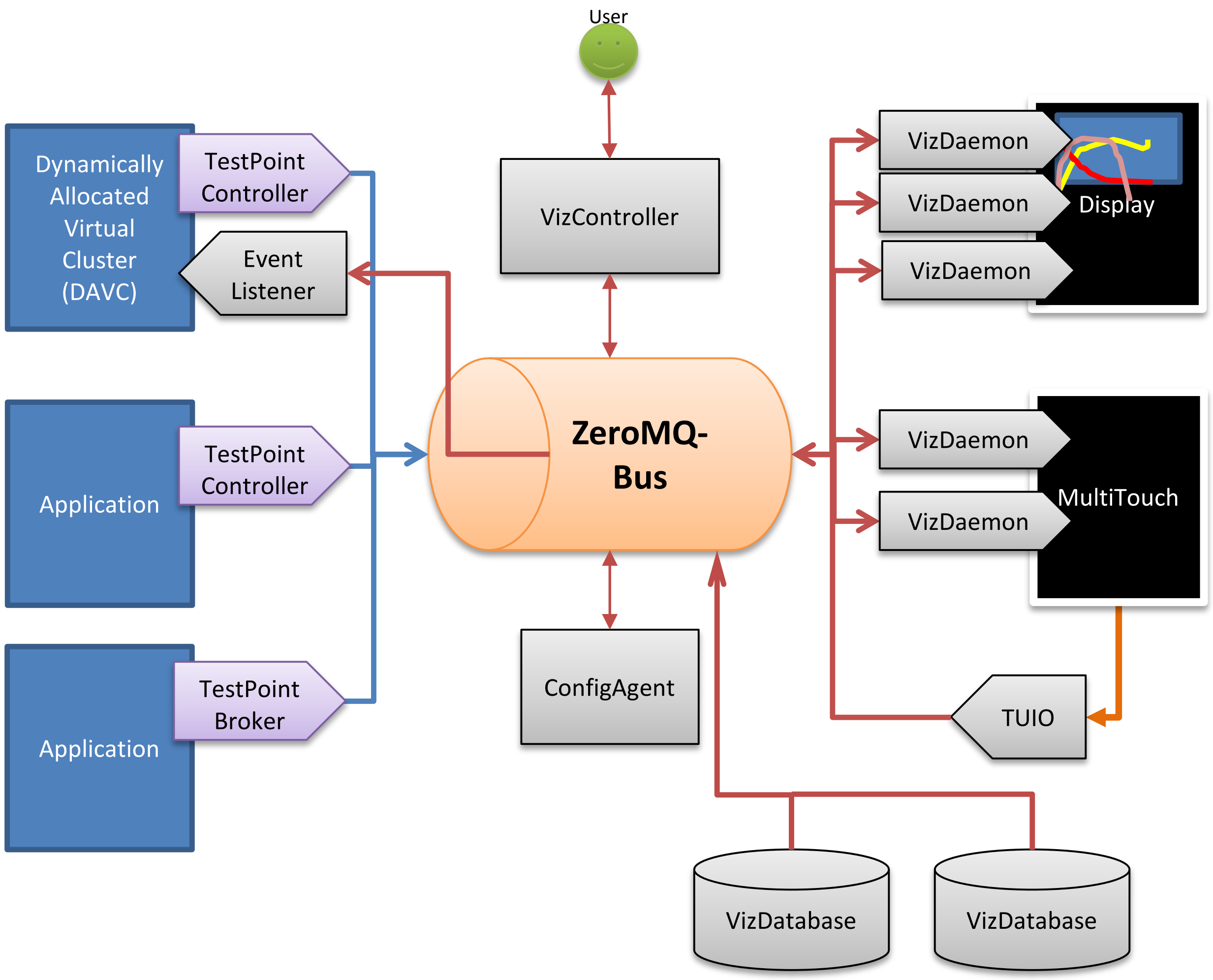

Fig. 2 ARL-VF with the TestPoint overview

The overall architecture of the ARL-VF is made up of the following components:

- **VizController**: Responsible for starting and stopping visualization processes by communicating with the VizDaemon. The controllers may also process events from the visualization daemons and share these events with other daemons. VizControllers may be instantiated as graphical user interfaces (GUIs) that are easy to control by people or may even be entirely scripted, such that the user starts the controller and a series of commands are sent in a scripted manner.

- **VizDaemon**: A set of daemons that receive input from the controllers and process data from the databases. Each VizDaemon services a single visualization tool and multiple daemons may be run on a single system.

- **VizDatabase**: A set of storage interfaces advertising content for use by the VizDaemon. The format of the content should be stored in a readily readable method by the VizDaemons.

- **ConfigAgent**: One or more of these agents run in the ARL-VF to provide a method to dynamically configure all the modules in the system. These agents rebroadcast discovery messages from other agents, enabling discovery of multiple modules through a single configuration agent.

- **EventListener**: Simple modules that subscribe to VizDaemon events. When running live simulations, these sort of events may be used to dynamically change something in the simulation.

- **TestPoint Controller/Brokers**: TestPoint controllers and brokers provide mechanisms for access to live data from a node using probes. These probes periodically gather data and publish them to subscribers using a similar format defined in Section 9.1.

- **Data Parsers**: A set of scripts used to convert data between well-known formats used by the visualization tools.

There may be any number of each of the above components on any host machine. For example, a laptop may be used during a demonstration to set up all the visualizers of a network science experiment across multiple screens. Then, during the demo, a presenter might right click on a node in one of the visualization daemons, generating an event and causing other VizDaemons to change their view. As the demo goes on, the presenter could bring up and down other visualizations using different controllers in the system. The methods of how each component connects with each other are described in the sections below.

When running with multiple simultaneous sources of data (live and or static), tags (see Section 4.1) are used to differentiate sources. For example, with 2 dynamically allocated virtual clusters (DAVCs), the tags may begin with “VizDaemon.DAVC1” and “VizDaemon.DAVC2”. These tags allow for differentiation of data that may otherwise appear very similar at the various modules.

In the ARL-VF, only VizDaemons and ConfigAgents bind to an Internet Protocol (IP) address and port. VizControllers and VizDatabases connect to these modules using the addresses contained in discovery messages. All over the wire messaging is done using Google protocol buffers and ZeroMQ. This is outlined in more detail in the following sections.

4. Messaging

In order to better integrate with TestPoint, a data collection framework developed by AdjacentLink, the ARL-VF will use both Google protocol buffers and ZeroMQ for all messaging in the system. The messages between components will be different than those used by the TestPoint, but easily convertible into TestPoint formats.

Because TestPoint is referenced several times in this section, the following are TestPoint specific endpoint definitions:

- **Probe**: Uses Python or C++ module to collect data from an isolated process.
- **Controller**: Manages all probes on a single system. A publish/subscribe discovery system is used for a controller to subscribe to the probes on the system.
- **Brokers**: Manages multiple controllers through a single broker. Commands sent to a broker are multicast to all controllers the broker is responsible for.
- **Recorders**: Subscribes to brokers or controllers to store data using SQLite for non-real-time retrieval.

While TestPoint also defines a format for configuration using extensible markup language (XML), we only use the over-the-wire message formats and allow modules within the ARL-VF to use their own configuration schemes.

4.1 Data Tag Format

The contents of this section are taken directly from the TestPoint Framework design document:[3]

> *TestPoint probe name format consists of a tree like naming convention where each element in the name tree is separated by a '. '. The more elements in a probe name, the more specific the probe name. This naming convention follows the ZeroMQ PUB/SUB subscription filter model. Client subscriptions are compared to published messages and those subscription names that match the published message name from the start of the published name through the length of the subscription name are delivered.*

For example, a probe named A.B.C.D may belong to a family of probes consisting of A.B.C.D, A.B.C.E, and A.B.C.F. Subscribing to a probe named A.B.C.D will only match that single probe. Specifying A.B.C will match A.B.C.D, A.B.C.E and A.B.C.F. Likewise, specifying just A will match all probes that start with A.

5. ZeroMQ Sockets

Table 1 summarizes the ZeroMQ sockets defined in Sections 5.1, 8, and 9. Unless otherwise noted in this report, all sockets use Transmission Control Protocol (TCP) for transport. The ZeroMQ sockets used by the ARL-VF are PUB/SUB and REQ/REP. When opening a ZeroMQ socket, it is necessary to specify the socket type (i.e., PUB). A PUB/SUB socket is used for one to many message distributions. The SUB end listens for data and the PUB ends sends data. Due to the nature of a PUB/SUB, there may be many subscribers and each message sent by a publisher is sent to all subscribers. A REQ/REP socket uses a pair-wise protocol between 2 nodes where a message is sent over the REQ end of the socket, and, when the REP end receives the message, it must respond. Until a response is received, the REQ end cannot send more data. In our system, the REP end always responds with 0 in a successful case and an error message for failure cases. Multiple REQ sockets can connect to the same REP socket.

Because the VizDaemon is the primary ZeroMQ server, we reuse the sockets for multiple purposes. Table 1 describes all sockets opened by the ARL-VF modules and the modules that connect to these sockets.

Table 1 ZeroMQ sockets defined in Sections 5.1, 8, and 9

Server	Server Socket	Client(s)	Client Socket	Description
ConfigAgent	SUB	VizDaemon, VizDatabase	PUB	Send Discovery message to ConfigAgent
ConfigAgent	XPUB	*	SUB	Broadcast all known Discovery messages
VizDaemon	REP	VizController	REQ	Send Command messages
		VizDatabase	REQ	Send static data
VizDaemon	XPUB	VizController	SUB	Send Event or Custom command messages
		VizDatabase	SUB	Send Probe Requests
VizDaemon	XSUB	VizController	PUB	Send Event messages
		VizDatabase	PUB	Send live data

For all REQ/REP sockets, the response should always be 0 when a message is successfully read and processed, and non-zero when an error occurs. Where possible, sockets in the table of the same type should be reused. For example, the VizDaemon binds to an XPUB socket and connects to the ConfigAgent using a PUB socket. Because ZeroMQ handles most socket functionality, implementations should use the same socket for listening and/or connecting to multiple hosts. In so doing, the VizDaemons will use a total of 3 sockets (XPUB, XSUB, and REP).

To simplify messaging with a topic header (for subscriptions), the ZeroMQ libraries provide "send_multipart" and "recv_multipart" functions. It is expected modules in this system will use these functions, putting the tag of the message as the first part and the serialized buffer as the second part.

Because of known limitations of pragmatic general multicast (PGM) within ZeroMQ, such as being unable to communicate across the loopback device, multicast is currently not being used in the system. This may change at a later date, potentially as a way for configuration agents to share information with each other.

5.1 Tag Prefixes

The following tags (Table 2) are used when sending data across modules. Each message type is described later this report.

Table 2 Tags

Tag	Subscriber	Description
*Source UUID	UUID owner	Used to send a publish message directly to a module. Each module must subscribe to messages for this own UUID.
VizDaemon.discovery	VizController, VizDatabase, ConfigAgent	Discovery messages from VizDaemons.
VizDatabase.discovery	VizDaemon, ConfigAgent	Discovery messages from VizDatabases.
VizController.cr	VizDaemon, ConfigAgent	Connection Request messages from VizControllers
<all other tags>	VizDaemon	Used when replying to probe requests. These tags may represent files or other data shared by a VizDatabase.

5.2 ZeroMQ Universal Resource Identifiers

Because of the use of different socket types in ZeroMQ, we use a modified uniform resource identifier (URI) from the base URI formatted used by ZeroMQ when sharing URIs within our system. Each ZeroMQ based URI will note the socket type in the scheme section of the URI. For example, a publish socket can be advertised using “pub://<address>:<port>”. Similar addresses are used for “sub://”, “req://”, etc.

6 Message Formats

There are 2 generic messages used by the ARL-VF: resource and data messages. Both types include some basic information and can be augmented by appending headers. This is done to allow adding new different headers for some modules without needing to change other existing modules.

Resource messages are used to share capabilities between modules. Capabilities must use universally unique identifiers (UUIDs) that are known to different modules publishing the capability. There are 3 capability types:

- **Source Capabilities**: These are capabilities about the module. For example, if the module is a MYSQL database front end, it would advertise the UUID for “mysql”.
- **Sink Capabilities**: These denote the kinds of inputs a module can handle. For example, if the module is an image view, it might be able to handle JPG, PNG, and GIF files. This module would note the associated UUIDs for these capabilities in its sink field.
- **General Capabilities**: All other capabilities.

Data messages are used for passing opaque blobs of data between modules. Each header added to a data message is additive. For example, the message could be a fragment of a file, in which case, both FileObject and FragmentObject headers would be added to the message. Messages are always noted as push or pull, denoting whether the message is pushing data or requesting it.

6.1 Resource Messages

Resource messages share basic information about the module and a list of zero or more source, sink, and general capabilities. The following fields are used in all resource messages:

- **Source_uuid**: A unique UUID for the sending module.
- **Name**: A string naming the module
- **Service_uris**: A list of ZeroMQ URIs. To see how these are formatted, see Section 5.2.

Any number of capabilities may be added to the resource message with the following fields:

- **UUID**: The UUID of the capability. All capability UUIDs must be unique such that any module can never interpret a UUID differently than another module.
- **Data**: Optional field for storing an opaque data object associated with the capability.

The following is an example of a resource message:

```
message Resource
{
  message Capability
  {
    required bytes uuid = 1;
    optional bytes data = 2;
  }

  required bytes source_uuid = 1;
  required string name = 2;

  repeated string service_uris = 3;

  repeated Capability source = 4;
  repeated Capability sink = 5;
  repeated Capability general = 6;
}
```

6.2 Data Messages

The fields in the DataObject message are the following:

- **MessageType**: Notes whether the message is a PUSH or PULL message. A PUSH message will contain some form of data. A PULL message is a request for data.
- **Source_UUID**: A unique UUID for the sending module.
- **Headers**: A list of data object headers used to describe the data contained in the message. See Section 7 for a list of headers included with the ARL-VF.
- **Data**: An opaque blob of data. This is only used for PUSH messages. PULL messages must have headers specifying the type of query used. Not all PUSH messages require setting this piece of data. Some data may be stored with a DataHeader.

Available data message headers are listed in the Appendix.

The following is an example of a data message:

```
message DataObject
{
  enum MessageType {
    PUSH = 1;
    PULL = 2;
  }

  message DataHeaders
  {
    required bytes type_uuid = 1;
    required bytes header = 2;
  }

  required MessageType type = 1;
  required bytes source_uuid = 2;

  repeated DataHeaders headers = 3;
  optional bytes data = 4;
}
```

7. Initialization

Figure 3 shows the structure used in initialization messaging.

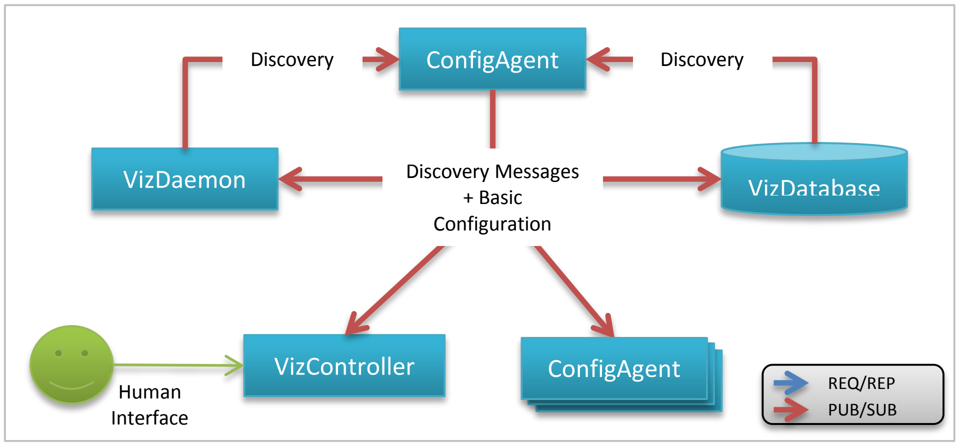

Fig. 3 Initialization messaging

The ConfigAgent component is included in the system to help simplify initial configuration. Upon initialization, modules connect to a configuration agent and publish resource messages (Section 9) with discovery headers, described in Section 13. These messages contain a service URI field to note the ZeroMQ sockets the module is binding to and also a set of tags used either to retrieve data. In most cases, VizDaemons are the only modules other than ConfigAgents that bind to sockets.

The VizDaemon and VizDatabase modules will use tag prefixes “VizDaemon.discovery” and “VizDatabase.discovery”, respectively, to differentiate between discovery messages from each module. This allows controllers to subscribe to discovery messages beginning with the tag “VizDaemon.”

Discovery messages must be sent periodically to the configuration agent. These messages act as heartbeats to prevent other modules from deciding a module is no longer there. An “exit” message is provided in the discovery message header to tell other modules when a module is exiting the system. A ConfigAgent may also send this message to force other modules to stop communicating with the exited module.

ZeroMQ is built to handle the case where a module connects to a socket before that socket is bound. For example, a controller may connect to the configuration agent before the agent is started. Once the configuration agent starts, ZeroMQ will properly finalize the connection. Likewise, if either module is suspended, ZeroMQ will properly restart that connection.

Finally, there may be multiple configuration agents in a system with multiple machines and enclaves. Each module may publish and subscribe to multiple configuration agents. However, configurations agents must not exchange messages with each other in order to avoid retransmission loops.

7.1 Asynchronous Routes

In a distributed system, it sometimes not possible for modules to connect to other modules due to not having an IP route to the destination machine. This can prevent VizController or VizDatabase modules from initiating a connection with a VizDaemon module. To work around this issue, modules can send special connection request messages over the PUB/SUB interface, requesting that the destination module (VizDaemon) instead initiate the connection. In this case, all transmissions usually sent over REQ/REP sockets are instead sent over PUB/SUB sockets, using the destination’s UUID as the prefix for the message so it is only received by the destination. The sequence diagram in Fig. 4 shows this interaction.

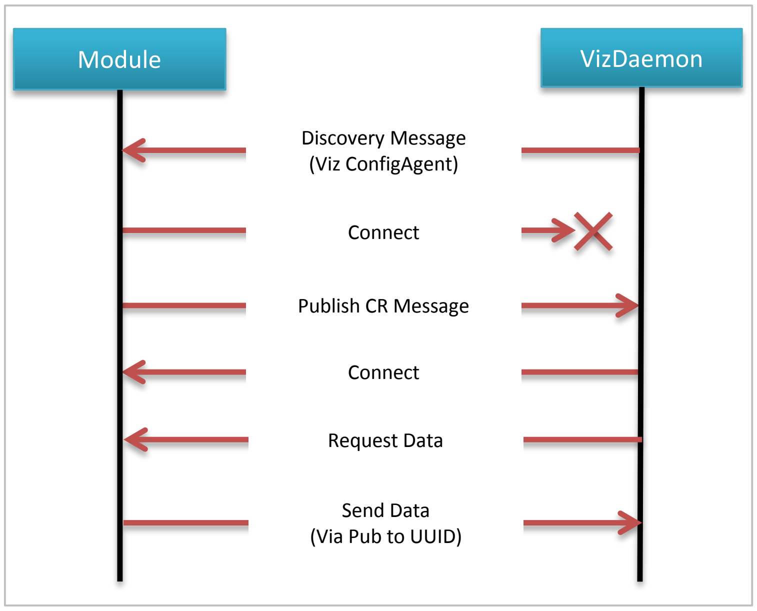
Module
VizDaemon
Discovery Message (Viz ConfigAgent)
Connect
Publish CR Message
Connect
Request Data
Send Data (Via Pub to UUID)

Fig. 4 Sequence diagram

An asynchronous route is detected when a connection attempt fails. When this occurs, the module attempting the connection instead binds to random ports for a PUB and SUB socket. Then a connection request header is added to a resource message. The receiving module will then connect to the provided PUB/SUB sockets.

In the case of VizDatabases, the ConnectionRequest is piggy-backed to the discovery messages, using the “VizDatabase.discovery” tag. In the case of a VizController, the connection request is sent using the tag “VizController.cr”.

Since all connections in the ARL-VF use a PUB/SUB message bug, all previous PUB/SUB message types will work as intended and are sent in the original manner.

7.2 Discovery Message Header

Resource messages with the discovery header have either TYPE_DISCOVERY or TYPE_EXIT. The former type notes that the module sending the message exists and the later type notes the module sending the message is exiting the system. The URIs needed to connect to the module are provided in the “service_uris” field in the parent resource message. The discovery message has the following fields:

- **Tags**: A list of all the available published by this module. The format for tags is specified in Section 6. Databases publish tags of files they can share. Daemons publish tags of files they can visualize.

Discovery messages with type set to TYPE_ERROR will have an Error message present in the response error member. The error message has the following field:

- **Description**: The description of the error.

The following is an example of a discovery message header:

```
message Discovery
{
  enum Type
  {
    TYPE_ERROR = 1;
    TYPE_DISCOVERY = 2;
    TYPE_EXIT = 3;
  }
  message Error
  {
    required string what = 1;
  }
  required Type type = 1;
  optional Error error = 2;
  repeated string tags = 3;
}
```

7.3 Connection Request Message Header

Connection request messages have the following fields:

- **Dest_uuid**: A list of UUIDs requested to connect to this module.
- **Service_uris**: The ZeroMQ URIs used to connect to the requesting module.

The following is an example of connection request message:

```
message ConnectionRequest
{
  repeated bytes dest_uuid = 1;
  repeated string service_uris = 2;
}
```

7.4 Use Cases

The following outline the various kinds of use cases.

VizDatabase initialization:

1) The module starts and looks at all available data. Each file is read in and the dot notation is reversed. For example, “test.1.txt” would become “txt.1.test”.

2) The VizDatabase creates a DiscoveryResponse message and fills out the "names" array with the reversed file names.
3) The DiscoveryResponse message is sent to the ConfigAgent using a REQ/REP socket. The ConfigAgent responds to the REQ with an empty string.
4) The VizDatabase then subscribes to the ConfigAgent's XPUB socket using "VizDaemon."
5) The ConfigAgent immediately pushes out all known discovery messages starting with "VizDaemon"
6) The VizDatabase then connects to each VizDaemon's SUB socket using XPUB.
7) For any subscription detected over the XPUB socket, the VizDatabase will push the requested data over the XPUB socket.

VizDaemon initialization:

1) VizDaemons subscribes to the XPUB socket of the ConfigAgent using the prefix "VizDatabase".
2) Upon detection of a new subscription to "VizDatabase", the ConfigAgent immediately pushes a DiscoveryReponse with all known VizDatabase entries.
3) The VizDaemon processes the DiscoveryResponse message, notes what data are available
4) The VizDaemon creates its own DiscoveryResponse message, filling in its ZeroMQ socket address, and the names of the data it can process, prepending "VizDaemon" to the names (i.e., VizDaemon.txt.run1) and pushes the message to the ConfigAgent
5) If a new DiscoveryResponse message arrives with new data this VizDaemon can process, it will resend a new DiscoveryResponse of its own to the ConfigAgent

VizController Initialization:

1) VizController subscribes to the XPUB socket of the ConfigAgent using the prefix "VizDaemon".
2) Upon startup, subscribe to "VizDaemon" messages from the ConfigAgent.

3) Receive any DiscoveryResponse messages and display available resources to the user.

ConfigAgent Initialization:

1) Open XPUB socket for DiscoveryResponse
2) Subscribe to any other ConfigAgent PUB socket to receive their DiscoveryResponse messages.
3) Push any hard-coded configuration over the XPUB socket.

VizDatabase receiving VizDaemon discovery (Asynchronous Route):

1) VizDatabase attempts to create a connection with the VizDaemon, which fails.
2) VizDatabase opens 2 sockets using random ports for PUB and SUB.
3) VizDatabase appends a ConnectionRequest header with the UUID of the VizDaemon to its discovery messages.
4) VizDatabase immediately sends discovery message.
5) Upon receiving connection request, VizDaemon connects to the VizDatabase.
6) All subsequent REQ/REP messages are instead sent over PUB/SUB using either the VizDaemon's or VizDatabase's UUID as the prefix for the publish message.

(The VizController asynchronous route case is identical to VizDatabase, but publish is sent using <VizDaemon UUID>.VizController.cr as the prefix.)

8. Controller/Daemon Interface

The purpose of the interface between VizControllers and VizDaemons is to initiate all available visualizations from a single interface. The visualization daemons communicate with the databases to see what available data are out there, determine which of that data each VizDaemon can parse, and then share that subset with the controller. The controller then provides the user the option to start these visualizations on the machine the daemons reside. Depending on the type of visualization, the interface between the controller and daemons allows simple commands, such as starting, stopping, and bringing down the visualization. The user is expected to handle more complicated interactions with the visualization tool by interacting directly with the tool. Figure 5 shows the controller/daemon interface.

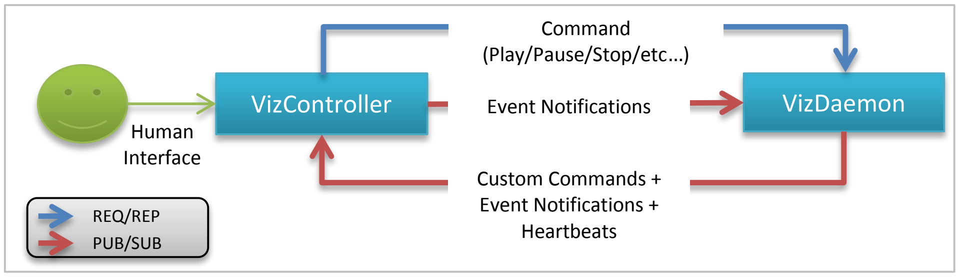

Fig. 5 Controller/daemon interface

Additionally, upon initiation of a visualization from the controller, the daemon may send the controller custom commands available to the user. These commands are specific to a given daemon and are not generalized to other daemons. Once the visualization is brought down, the controller forgets about these commands. An example may be a visualization component offering a way to manipulate the current view shown, providing 4 different views and the command to switch between each view.

Beyond starting and stopping visualizations, the controller/daemon interface also provides a bus for sharing events. An event may be a mouse click by the user or something triggered by a live data stream. The interface allows the VizDaemons to capture events and share them with any active controllers. The controllers, in turn, may share these events with the user and/or pass the events to other connected daemons. An example of how this could be useful is to dynamically alter a visualization instance when an event occurs in a separate visualization.

As mentioned in the Initialization section, the VizDaemons bind to sockets and the VizControllers connect to these sockets. The daemons use an XPUB interface to detect new subscriptions and upon detection provide both custom commands and any previous events the daemon chooses to. It is expected the capabilities of the daemon are already known by the controller through the ConfigAgent and updates to these capabilities are shared through the ConfigAgent interface (see Section 5.1).

The VizDaemons periodically send their discovery messages across their subscription channel with the prefix “VizDaemon.discovery” as a form of heartbeat. This allows other modules to quickly detect if a daemon goes down and also allows finding daemons on the localhost by simply connecting to any daemon on a well-known port over the loopback interface.

8.1 Control Message Format

This section describes the Google protocol buffer definition for the messages between VizControllers and VizDaemons. The following is an example of a control message:

```
message ControlMessage
{
  enum MessageType
  {
    TYPE_ERROR = 1;
    TYPE_COMMAND = 2;
    TYPE_CUSTOM = 3;
    TYPE_EVENT = 4;
  }
  enum CommandType
  {
    CMD_ERROR = 1;
    CMD_START = 2;
    CMD_STOP = 3;
    CMD_PAUSE = 4;
    CMD_GOTO = 5;
    CMD_DESTROY = 6;
    CMD_CUSTOM = 7;
  }
  enum EventType
  {
    EVENT_ERROR = 1;
    EVENT_COMMAND = 2;
    EVENT_MOUSECLICK = 3;
    EVENT_OTHER = 4;
  }
  message Error
  {
    required string description = 1;
  }
  message Command
  {
    required CommandType type = 1;
    optional string args = 2;
  }
  message Custom
  {
    repeated string commands = 1;
    repeated string descriptions = 2;
  }
  message Event
  {
    required EventType type = 1;
    optional string info = 2;
    optional bytes buf = 3;
  }
  required string tag = 1;
  required uint32 version = 2;
  required MessageType type = 3;
  optional Error = 4;
  optional Command = 5;
  optional Custom = 6;
  optional Event = 7;
}
```

The fields in this message are the following:

- **Tag**: The tag assigned to the probe data.
- **Version**: The version of the control message.
- **Type**: Notes whether this is an error, command, custom, or event type message

ControlMessage messages with type set to TYPE_ERROR will have an error message present. The error message has the following fields:

- **Description**: Description of the error that occurred.

ControlMessage messages with type set to TYPE_COMMAND will have a command message present. The command message has the following fields:

- **Type**: The type of command being sent as noted from the CommandType enum.
- **Args**: Additional arguments to pass along with the command. If this is a custom command, the args value contains the entire custom command.

ControlMessage messages with type set to TYPE_CUSTOM will have a custom message present. The custom message has the following fields:

- **Commands**: An array of commands to send to the VizDaemon.
- **Description**: An optional description of what is being done (human readable).

ControlMessage messages with type set to TYPE_EVENT will have an event message present. The event message has the following fields:

- **Type**: The type of command being sent as noted from the EventType enum.
- **Info**: Description of the event

- **Blob**: Opaque structure to hold event information. Some events may be contained in their own protocol buffers, which is discussed in later sections.

8.2 Use Cases

The following are the controller use cases.

Controller connects to daemon:

1) Controller creates SUB socket and calls ZeroMQ connect.
2) Daemon detects new SUB connection on XPUB interface, pushes out custom commands.
3) Daemon also pushes out any events that may have occurred.

Controller starting visualization:

1) Controller sends a start signal to the daemon for a given file over the REQ socket.
2) Daemon receives start signal and retrieves the file or stream from a database over the REP socket.
3) Daemon starts the visualization instance with the given file over the REQ socket.

Controller sending custom command:

1) Controller sends a custom command signal to the daemon
2) Daemon either starts a new visualization or modifies and existing visualization.

Daemon receives event (right mouse click example):

1) User right clicks on node in visualization.
2) Daemon detects the right mouse click and sends event message to controller over the XPUB socket.
3) Controller receives right mouse click event and pushes it to other connected daemons.
4) Other daemons process the right mouse click as configured to possibly change how they display data.

8.3 Event Listeners

Event listeners are special modules in the ARL-VF that only listen to events, but otherwise do not interact with other modules. Because of the simplicity of these modules, all they require is to subscribe to the ConfigAgents to determine where the VizDaemons are located and then to subscribe to the event messages from each VizDaemon.

9. Daemon/Database Interface

Figure 6 shows the daemon/database interface.

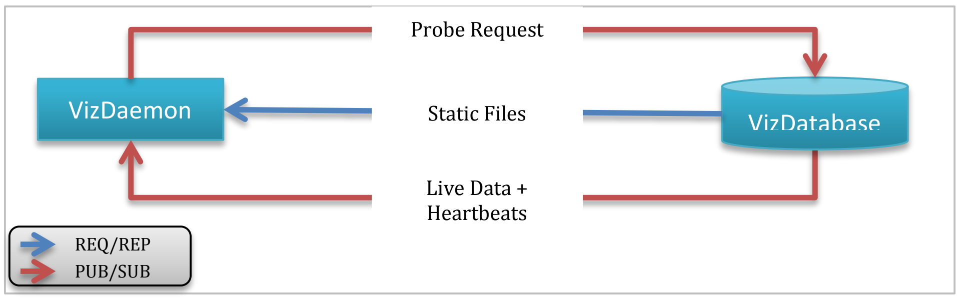

Fig. 6 Daemon/database interface

Though the message format is different, the interactions between VizDaemons and VizDatabases are modeled after the probe interactions in TestPoint. However, unlike TestPoint, which is intended for live data, this interface additionally offers a bus for static data exchange. In the live case, TestPoint acts as a VizDatabase, providing data to the visualization daemons. In the static case, the VizDatabase stores and gathers data from either the file system or a relational database.

The exchange of available files and probes from the VizDatabase to the VizDaemons is described in Section 5.1, where discovery messages contain a "tags" field with a list of files available in the VizDatabase. Actual files are exchanged when a VizDaemon subscribes to and requests a given probe data prefix (see Section 4.1). For example, if the VizDaemon subscribes to "Emane.Run1", all files/probes using that prefix will be sent to the visualization daemon. Because static file names are listed in reverse dot notation ("txt.1.run" instead of "run.1.txt"), subscriptions can also be generic to a file type ("txt.").

Because the VizDaemons bind to ZeroMQ sockets, the VizDatabase will connect to all known daemons upon learning about them via the ConfigAgent. VizDatabases and VizDaemons communicate using 2 sockets. The first is a REQ/REP socket for receiving probe requests and responding with static data, where the VizDaemon has the REQ end and VizDatabase the REP end. The second is a PUB/SUB socket for pushing live data to all subscribers. In order to receive ongoing live data, the VizDaemon must subscribe to the probe prefix of the VizDatabase's PUB socket as well as send a probe request.

The VizDaemons periodically send their discovery messages across their XPUB channel with the prefix "VizDaemon.discovery" as a form of heartbeat. This allows other modules to quickly detect if a daemon goes down and also allows finding daemons on the localhost by simply connecting to any daemon on a well-known port over the loopback interface.

As mentioned in 7.4, Probe tags for files are written in reverse dot-order of file names. For example, "test.1.txt" would become "txt.1.test".

9.1 Probe Message Header

Probe message headers are sent encapsulated within a data object message (see Section 10). These may be either PULL or PUSH messages, where PULL messages originate from VizDaemons to request a certain tag and PUSH messages originate from VizDatabases to push files or live data streams. The fields in this message are the following:

- **Type**: The message type used to identify whether the probe report contains data or an error message.
- **Tag**: The tag associated with the probe.
- **Index**: The index of the probe. Each probe instantiated by the same controller is assigned a unique sequential index.
- **Timestamp**: The timestamp of the report.
- **Request_uris**: If this is a PULL message, this field will list the URIs the sender is listening on to receive the response. The URIs are formatted as specified in Section 8)

When static files are sent, the FileObjectHeader must be included in the DataObject message. When live data are sent, the TimedDataStreamObjectHeader must be included in the DataObject message.

ProbeReport messages with type set to TYPE_OK are standard messages. Messages with TYPE_STOP tell the VizDatabase to stop sending a live stream with the associated tag. Messages with TYPE_ERROR will have an error message present in the report error member. The Error message has the following field:

- **Description**: The description of the error.

The following is an example of a probe message:

```
message ProbeReport
{
  enum MessageType
  {
    TYPE_ERROR = 1;
    TYPE_OK = 2;
    TYPE_STOP = 3;
  }

  message Error
  {
    required string description = 1;
  }

  required MessageType type = 1;

  required string tag = 2;
  optional uint32 index = 3;
  optional uint64 timestamp = 4;

  repeated string request_uris = 5;

  optional Error = 6;
}
```

9.2 Use Cases

The following are the VizDaemon use cases.

VizDaemon requests static data:

1) VizDaemon sends a PULL message with a ProbeReport header across existing XPUB socket for a given tag. The request_uri field is filled out with the daemon's REP socket URI for the database to connect to and send the data.

2) The VizDatabase create a PUSH message with a ProbeReport header for all files matching the requested tag prefix.

3) Each ProbeReport is sent to the requesting VizDaemon using the specified REQ/REP socket.

4) After each transmission, the VizDaemon responds to the database with a 0 to indicate the file was received or –1 otherwise.

VizDaemon requests live data:

1) VizDaemon subscribes to the tag on the PUB/SUB interface
2) VizDaemon sends a PULL message with a ProbeReport header across existing XPUB socket for a given tag. The publish field is filled out with the daemon's SUB socket URI for the database to send the data.
3) The VizDatabase create a PUSH message with a ProbeReport header for all live data matching the requested tag prefix.
4) If the database has no historical data to send, the PUB/SUB socket is connected to as described in steps 3 and 4 in the previous use case.
5) The database begins publishing the data specified by the tag over the PUB/SUB interface

9.3 TestPoint Translation Layer

The interface specified by TestPoint primarily involves connecting to its controller or broker nodes. These nodes may have several probes associated with them. Since the method of connecting to these controllers is different than how the ARL-VF binds and connects sockets, an intermediate process will be necessary to act as a VizDatabase to the ARL-VF and act as a client to a controller or broker in TestPoint. However, because the message formats are identical or very similar, the translation should be straightforward.

10. Conclusion

The initial iteration of the ARL-VF has laid much of the groundwork for future NSRL visualization capabilities, and has proven the concepts of the system. Future work will include continued improvements to performance and ease of use as well as an increased set of visualizations, which will be developed using a wide array of programming languages and graphics tools. Our intent is to develop an approach that fosters a community of contributors through the use of open standards, so that researchers may select the proper view or views of their results without the burden of implementation details for the extraction of available data.

11. References

1. Extendable Mobile Ad-hoc Network Emulator (EMANE) [accessed 2015], http://www.nrl.navy.mil/itd/ncs/products/emane.

2. Common Open Research Emulator (CORE) [accessed 2015], http://www.nrl.navy.mil/itd/ncs/products/core.

3. Galago S. TestPoint Data Collection Framework Rev 1.3. Adjacent Link LLC. (written on contract to ARL).

4. NRL Scripted Display Tool 3D (SDT3D) [accessed 2015], http://www.nrl.navy.mil/itd/ncs/products/sdt.

5. NSRL Public web site [accessed 2015], http://www.arl.army.mil/nsrl.

INTENTIONALLY LEFT BLANK.

Appendix. Data Message Headers

The following headers are included in the system. Users may choose to add more headers, but these may or may not be parsable by modules within the ARL-VF:

```
message MultipartObjectHeader
{
  required bytes uuid = 1;
  required uint32 object_num = 2;
  required uint32 total_objects = 3;
}
```

A multipart object header is used when a blob of data cannot be sent as a single message, but is not a fragment:

- **UUID**: Unique identifier for this message. All parts of this message must use the same UUID.
- **Object_num**: The object number in the multi-part message. For example, this is the 5th of 10 objects.
- **Total_objects**: The total number of messages in this multiple message.

```
message FileObjectHeader
{
  optional string SHA1 = 1;
  optional string fname = 2;
  optional string mime_type = 3;
}
```

Used to note that this object is a file.

- **SHA1**: The SHA1 hash of the file.
- **Fname**: The name of the file.
- **Mime_type**: If this is a well-known file type, the mime type of the file.

```
message TimedDataStreamObjectHeader
{
  required bytes uuid = 1;
  optional string name = 2;
  required uint64 start_time = 3;
  required uint64 end_time = 4;
  optional bool finish = 5;
}
```

This object is part of a stream and may contain multiple messages. Each message in the stream must use the same UUID and the start and end times must not overlap.

- **UUID**: Unique identifier for this message. All parts of this message must use the same UUID.
- **Name**: Optional name identifying this message stream.

- **Start_time**: The start time of this message in seconds.
- **End_time**: The end time of this message in seconds. This is not an offset of the start time and must be greater than the start time.
- **Finish**: An optional token noting that the stream is finished. Modules must not send another message with this UUID after this token is set or receiving modules may drop the message.

```
message StringObjectHeader
{
  required string info = 1;
}
```

This object is a string, contained in the “info” field:

```
message FragmentObjectHeader
{
  required bytes uuid = 1;
  required uint32 fragment_num = 2;
  required uint32 total_fragments = 3;
}
```

A fragment object is a single blob of data that is broken into parts. For example, a very large file may be fragmented into smaller chunks prior to sending. A recipient may choose to reassemble the blob of data once all fragments are received by appending the opaque buffers in order. Another header, such as a FileObjectHeader, is used to note what type of data are contained in the total buffer.

- **UUID**: Unique identifier for this message. All parts of this fragment must use the same UUID.
- **Fragment_num**: The fragment number in the series of fragments.
- **Total_fragments**: The total number of fragments used for this blob of data.

```
message TableObjectHeader
{
  message TableCell
  {
    optional string column_name = 1;
    optional string row_name = 2;
    optional bytes cell_data = 3;
  }

  repeated TableCell cells = 1;
}
```

This object is stored in a table, such as a structured query language (SQL) table. These messages contain their own opaque data objects and do not use the data field in the DataObject message.

- **Column_name**: The name of the column of this message.
- **Row_name**: The name of the row of this message.
- **Cell_data**: The contents of the cell.

List of Symbols, Abbreviations, and Acronyms

ARL	US Army Research Laboratory
ARL-VF	US Army Research Laboratory Visualization Framework
CORE	Common Open Research Emulator
CPU	central processing unit
DAVC	dynamically allocated virtual cluster
EMANE	Extendable Mobile Ad hoc Network Emulator
GUI	graphical user interface
IP	Internal Protocol
NRL	US Naval Research Laboratory
NSRL	Network Science Research Laboratory
OS	operating system
PGM	pragmatic general multicast
RF	radio frequency
SQL	structured query language
TCP	Transmission Control Protocol
URI	uniform resource identifier
UUID	universally unique identifier
XML	extensible markup language

1	DEFENSE TECHNICAL
(PDF)	INFORMATION CTR
	DTIC OCA
2	DIRECTOR
(PDF)	US ARMY RESEARCH LAB
	RDRL CIO LL
	IMAL HRA MAIL & RECORDS
	MGMT
1	GOVT PRINTG OFC
(PDF)	A MALHOTRA
1	US ARMY RESEARCH LAB
(PDF)	RDRL CIN
	A KOTT
6	US ARMY RESEARCH LAB
(PDF)	RDRL CIN T
	W DRON
	M KEATON
	A TOTH
	J HANCOCK
	M AGUIRRE
	B RIVERA